Customer No. 27871

OIPE IAP42 MAR 2 8 2008 PATENT & TRADEMARK OFFICE

ZFW

IN THE UNITED STATES PATENT & TRADEMARK OFFICE

Appl. No.:	**10/787,094**	Art Unit:	**2616**
Applicant:	**BOISVERT, Eric et al.**	Examiner:	**HAILU, Kibrom T.**
Filed:	**February 27, 2004**	Docket No.:	**101783/00006**
Title:	**MASSIVELY REDUCED INSTRUCTION SET PROCESSOR**		

U.S. Patent & Trademark Office
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

TRANSMITTAL OF CERTIFIED COPY OF PRIORITY DOCUMENT

Sir:

In support of Applicant's claim for foreign priority pursuant to 35 U.S.C. 119(b) submitted herewith is a certified copy of Canadian Patent Application No. 2,443,347 filed on September 29, 2003.

Respectfully submitted,

[illegible]

Brett J. Slaney
Agent for Applicant
Registration No. 58,772

Date: March 28, 2008

BLAKE, CASSELS & GRAYDON LLP
Suite 2800, P.O. Box 25
199 Bay Street, Commerce Court West
Toronto, Ontario M5L 1A9
CANADA

Tel: 416-863-2518
BS/dm
21752237.1

Office de la propriété intellectuelle du Canada

Un organisme d'Industrie Canada

Canadian Intellectual Property Office

An Agency of Industry Canada

Bureau canadien des brevets

Certification

Canadian Patent Office

Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification and Drawings, as originally filed, with Application for Patent Serial No: **CA 2443347**, on September 29, 2003, by **PLEORA TECHNOLOGIES INC.**, assignee of George Chamberlain, Alain Rivard and Eric Boisvert, for "Massively Reduced Instruction Set Processor".

L. Lachance

Agent certificateur/Certifying Officer

February 29, 2008

Date

Canada

(CIPO 68)
31-03-04

OPIC CIPO

ABSTRACT

This invention is directed to a method and apparatus for providing low, predictable latencies in processing IP packets. The apparatus provides a specialized microprocessor or hardwired circuitry to process IP packets for video communications and control of the video source without an operating system. The method relates to operation of a microprocessor which is suitably arranged to carry out the steps of the method. The method includes details of operation of the specialized microprocessor.

Massively Reduced Instruction Set Processor

FIELD OF INVENTION

This invention relates in general to microprocessors, and in particular, to a microprocessor used for data communications.

BACKGROUND OF THE INVENTION

Over the last few decades Internet Protocol (IP) communications have become the dominant form of electronic communication. IP communications allow the use of a wide array of different protocols. To simplify data handling and routing, the protocols are arranged in a stack and the "lowest-level" protocols encapsulate the higher-level protocols. This encapsulation allows the idiosyncrasies of the higher level protocols to be hidden from the routing functions and further allows the partitioning of the analysis of the data.

In stand-alone devices, also known as embedded products and embedded devices, embedded computers are typically used to perform the encapsulation and de-encapsulation to send and receive the data respectively. An embedded computer is characterized as having a general purpose CPU, with associated memory. The computer runs an Operating System (OS), such as embedded Linux. The protocol processing is handled by the OS and application software is provided that runs on top of the OS to handle the communications functions and other tasks that are required.

This architecture is analogous to what is provided on general purpose computers (PCs) and workstations. Using the same processes to handle the communications in the embedded device as are used on general purpose computers is natural since IP communications was first performed only on general purpose computers and later migrated to embedded devices.

However, different from general purpose computers, embedded devices only have limited resources and are highly cost sensitive. The processor that can be employed in an embedded computer is often very limited in performance due to cost, space, and power consumption constraints. As a result an embedded device often cannot be cost effectively IP enabled for high-bandwidth devices.

To handle multiple tasks a real-time operating system (RTOS) is often employed which provides the abilities to respond to system requests in a very short period of time. Even with this, applications such as high performance image delivery for machine vision find the level of latency and the variation in the latency associated with the delivery of the video to be unacceptable. Further, when OS-based embedded devices are pushed to their limits they can become unreliable with deadlocks that freeze the device.

It is obvious that the above implementations do not address the requirements for protocol processing on a device, such as a high-speed electronic video camera or other high-bandwidth device. Therefore there is a need for a method and apparatus capable of processing IP packets with low, consistent latencies that are suitable for delivering video over an IP network.

SUMMARY OF THE INVENTION

This invention is directed to a method and apparatus for providing low, predictable latencies in processing IP packets. The apparatus provides a specialized microprocessor or hardwired circuitry to process IP packets for video communications and control of the video source without an operating system. The method relates to operation of a microprocessor which is suitably arranged to carry out the steps of the method. The method includes details of operation of the specialized microprocessor.

In accordance with one aspect of this invention, a massively reduced instruction set processor (mRISP) is disclosed which is a tiny embedded soft processor tailored for processing communication protocols in accordance with the method disclosed herein. In a preferred embodiment, this processor has only two instructions and some optional registers performing basic functions, such as arithmetic and logical functions, and specialized functions like Program Counter, Timers, IP Checksum and DMA. The soft implementation of the mRISP is realized since it is fully configurable upon construction through synthesis of a register transfer level (RTL) representation of the design by specifying the registers and the features required in the implementation. The processor that is created from the synthesis is tailored for a specialized task, such as data communications.

The two IPP instructions are LOAD and MOVE which are the minimal instructions necessary for a processor. Some macros are built over these two instructions in conjunction with registers to add some other basic functionality like JMP, CALL and RET. The macros are used in the compiler for the instruction set for the IPP, and are built solely using the LOAD and MOVE instructions.

The core is maximally optimized for a 16-bit data bus and a 32-bit instructions bus, although it can be configured for wider or narrower bus widths. In 16-bit data mode, bytes can be swapped for single byte access and operation. The 32-bit instructions bus, separated from the data bus, allows the timing to be reduced to only one clock cycle for a LOAD and two clock cycles for a MOVE. An extra clock cycle is added to the timing on a jump in the program counter.

For slow external memory fetching or for any other specific reasons, external logic can be added to control the HOLD input signal and holds the processor for a required number of clock cycles. In addition to that, specialized waiting functions, if required and activated, can hold the processor until an expected event occurs.

With such a processor, IP packets can be processed at significantly higher rates, with lower, consistent latencies, than can be accomplished using a general purpose microprocessor

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is the preferred INVENTION Embodiment.
Figure 2 is a description of the mRISP State Machine
Figure 3 is a description of the Data Path
Figure 4 is a description of the checksum register
Figure 5 is a description of the MOVE register
Figure 6 is a description the LOAD register
Figure 7 is a description of the General Purpose Register A

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In the following detailed description of the embodiments, reference is made to the accompanying drawings, which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that structural, logical and electrical changes may be made without departing from the spirit and scope of the present inventions. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present inventions is defined only by the appended claims.

The mRISP implements the CPU with separate data and program memory bus, generally known as a Harvard memory architecture. The mRISP program memory bus is a 32-bit wide used to fetch instructions from memory. The mRISP data bus is a 16-bit wide used to move 16-bit word data from user memory, internal registers or program memory to user memory and internal registers. The external user memory bus may be connected to memories or peripherals.

Instruction Set

The mRISP instruction set is massively reduced to only two instructions. The first one is the instruction MOVE which one moves data from a source address to a destination address. The only other one necessary for a functional CPU is the instruction LOAD which one can initialize memory and registers to a proper value from the program memory.

The 32-bit instruction contains only one bit to decode the opcode. On an instruction MOVE, 14-bit is dedicated for the source address and another 14-bit is for the destination address, leaving 3 bits unused. On an instruction LOAD, 14-bits are used for the destination address and 16-bits for the constant word to load, leaving 1 bit unused.

The MSB bit of the addresses (source and destination) is used to select between the external user memory region and the internal registers region. The LSB bit of the addresses (source and destination) is used to decode if data bytes swapping has to be done. Thus 12 bits out of 14 bits are available to user memory and peripheral. The external memory address is in word (16-bit).

Figures 5 and 6 provide a bit by bit description of the MOVE and LOAD instructions.

Figure 21 provides an implementation of the instruction formatter with the opcode decoder, address detector and byte swapping detector.

Internal Registers and Functions

The mRISP has two general-purpose registers (REG_A and REG_B) and some dedicated registers performing specific functions (i.e. features). Those functions may be any combination or number of arithmetic (increment, adder), logical (AND, OR, XOR), comparators, timers, DMA, interrupts and program counter functions. The arithmetic and logical registers use the general-purpose registers with the mask register as inputs. Their values are constantly updated as general-purpose registers change.

Two registers are specifically designed to process Internet Protocols. The first one (CSUM) is useful to compute Internet Protocols checksums. The method used to compute the IP checksum is the 16-bit one's complement sum of the corresponding data. Each time a write is done into the CSUM register, the 16-bit one's complement addition is computed from the previous value and the written value. When all the data to be included in the checksum has been written in this register, the read of this register gives the 16-bit one's complement sum by inverting the present value. A read resets the CSUM register to zero, ready for another computation. By filling the checksum field(s) in the IP header(s) with a magic number, the checksum can be serially performed as the data is being packetized. One byte is added at the end of the packet with the appropriate data necessary to make the magic number in the header field correct.

The second register (DMA) is used to move multiple data from one location to another one within three instructions. When one location is an internal register, its address is not incremented, enabling the capability to send consecutive data in memory into one special register or initialize consecutive data in memory with one register's value. In conjunction with CSUM, it is easy to quickly compute Internet Protocols checksums with only a few instructions.

Comparators between REG_A and REG_B are constantly computed. Two flags are necessary to do all comparison (equal '==', not equal '!=', less than '<', greater than '>', less than or equal '<=' and greater than or equal '>='). The first one's is the "A Equal B" flag (eq) and the second one's is the "A Greater than B" flag (gt). Those flags are used in conjunction with the Program Counter (PCNT) to enable conditional jumps. The descriptions of the General Purpose registers, Program Counter, Return Register, Mask

Register, Wait Register, Timer 0 Register, Timer 1 Register, Checksum Register and DMA Register are provided in Figures 7 – 15 respectively.

Figure 22 provides a description of the Internal Registers and Functions.

Program Counter and Return Registers

The Program Counter register (PCNT) is cleared to zero on reset and is incremented by one on last cycle of every instruction (when *prd* is high). It always points to the next instruction during the processing of the current instruction. A jump in the program memory can be accomplished by writing the new instruction's address in the Program Counter register. The jump can be conditional or not, depending on the state of the comparator flags (*eq* and *gt*) and the setting of the three flag bits (IE, IG and IN) in the Program Counter Register.

A CALL instruction can be accomplished by writing in the Program Counter register the sub-routine's address and by setting the flags to IE=0, IG=0 and IN=1. In this case, the Return register (RETA) loads the Program Counter's value at the same time the jump is done. Later, on a RET instruction (by moving RETA's value into PCNT register), the mRISP can resume fetching instructions on the next one's after the CALL instruction. The stack is hardware and its depth is configurable at the synthesis. The stack is structured as a LIFO (Last In First Out). On a CALL instruction, the Program Counter's value is pushed in the LIFO and on a RET instruction, the value to write into the Program Counter is pulled from the LIFO.

Figure 16 summarizes the jump and the call conditions when writing in the Program Counter.

Event Handling

The mRISP allows up to 16 events, which can be generated from any of the two sources: external hardware interrupts or internal events. The internal events may come from timers, real-time timer and watchdog logic. All events are completely handled by software and no event can interrupt the execution of the program. The software must verify itself in the WAIT register if an event occurred. The software can put the processor in the sleep mode by setting in the WAIT register the bit(s) of the corresponding event(s) it want to be waked up.

According to Figure 12, writing one in the event 'X' bit of the WAIT register, sets to one the corresponding "SET" signal (*wait_x_set*) and, at the same time, sets to one the global signal *wait*. Then the processor goes in the sleep mode and waits for the event X.

When this event occurs (*event_x* goes to one), the corresponding "EVENT" signal (*wait_x_evt*) is set to one. One clock cycle later, this signal clears the SET signal (*wait_x_set*) and the global signal *wait*. Thus the processor resumes its operations.

The software has the responsibility to clear the EVENT bit and to retrieve which event waked up the processor if more than one bit has been set in the wait register. By reading the WAIT register, the software reads all the EVENT bits (*wait_?_evt*) and also clears most of the bits (timer event bits are only clear by writing in the corresponding TIMER register).

Figure 20 provides an implementation of the Event Block.

Macros

Macros are added to instructions that are interpreted by the compiler. These make the mRISP easier to program and makes the resulting assembly code more understandable and maintainable. These are built over the two instructions in conjunction with registers. For example the JMP macro, which one is used to jump in another part of the program, is in fact a LOAD instruction with the destination address equals to the Program Counter register's address and the constant data equals to the address to jump in the program memory.

Figure 18 provides a possible set of macros that could be used.

Data Path

For each instruction, a 16-bit data word is transferred from one location to another one. The source may be from the program memory (on a LOAD), from one of the internal registers or from the user memory (on a MOVE). The destination may be either one of the internal registers or the user memory.

The higher byte and the lower byte in the data may be swapped together when only one of the location address is odd (bit 0 is high). This is very useful to reverse the byte ordering since Internet Protocols are big-endian and the mRISP is little-endian.

Figure 3 describes the data path.

State Machine

The mRISP state machine synchronizes internal and external control signals to provide efficient timing. The LOAD instruction takes only one clock cycle and two clock cycles for a MOVE. An extra clock cycle is added to the timing on a jump in the program counter.

Figure 2 provides a diagram of the mRISP State Machine. The State Machine has only four states. The RESET state is reached whenever the signal *rst_n* is asserted. At the first cycle where *rst_n* is de-asserted, the state machine goes to the FETCH32 state.

The FETCH32 state decodes the instruction presented on the *pdata_in* bus. Depending on the value of the opcode and the signals *hold* and *wait*, the next state can be WAIT_ACK, JUMP or FETCH32 again. The signal *wait* is used in this state to keep the processor waiting for an event, defined previously by writing in the WAIT register. During this waiting, no instruction fetching, no writes and no reads are performed. In the FETCH32 state, the signal *hold* has the same effect as the signal *wait* but it is generated by external logic. The reason for its assertion may be that data from the program memory is not ready due to slow memory, that the write from the previous instruction into external memories takes more than one clock cycle or for any other reasons. If the signals *wait* and *hold* are not asserted and the opcode is MOVE, a read is performed from the source address and the next state is WAIT_ACK. Otherwise the instruction LOAD is performed. The constant data contained in the instruction is written to the destination address. If the destination address is the Program Counter Register (PCNT) and the flag indicates an unconditional jump or a true conditional jump (signal *jump* is asserted), the State Machine goes in the JUMP state. Otherwise, it stays in the same state, ready for the next instruction.

The WAIT_ACK state waits for the read data from the source address to be ready. If it's not, the external logic must keep the signal *hold* asserted until data is ready. When it is ready, the State Machine comes back in the FETCH32 state unless the destination address of the MOVE instruction was the Program Counter Register (PCNT) and the flag indicated an unconditional jump or a true conditional jump (signal *jump* is asserted). In this last case, the next state is going to be JUMP.

The JUMP state is an idle state where the cycle is used only to fetch the instruction pointed by the new address loaded in the Program Counter Register.
The State Machine comes back in the FECTH32 state unless the external logic keeps the signal *hold* asserted for any reason.

A cycle by cycle representation of the State Machine with different cases is provided in Figure 19.

It is to be understood that this description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reviewing the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

The embodiment(s) of the invention described above is (are) intended to be exemplary only. The scope of the invention is therefore intended to be limited solely by the scope of the appended claims.

CLAIMS

1. Apparatus providing a specialized microprocessor or hardwired circuitry to process IP packets for video communications and control of a video source.

2. A method for operation of a specialized microprocessor or hardwired circuitry to process IP packets for video communications and control of a video source.

Figure 1

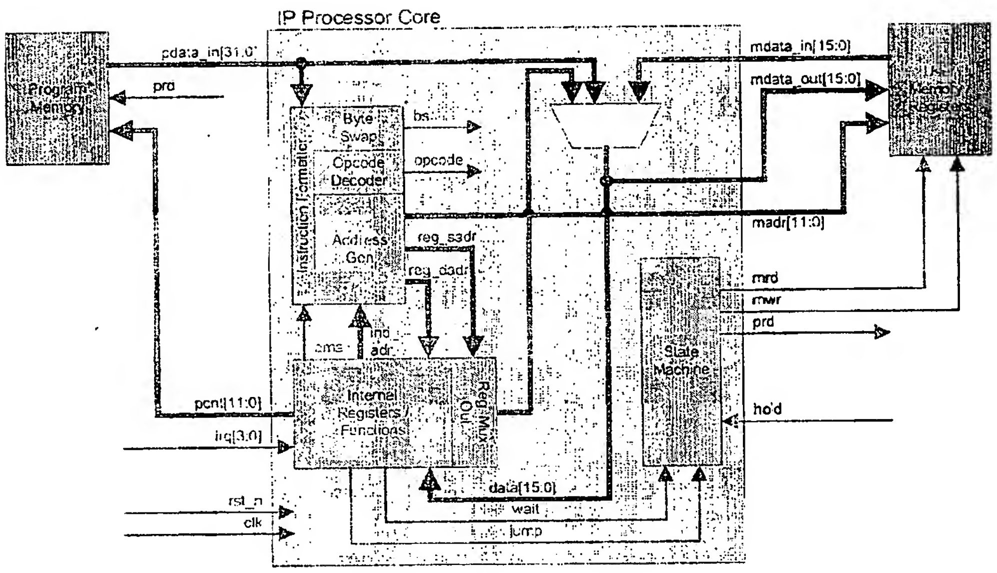

IP Processor Core
Program Memory
pdata_in[31:0]
prd
Byte Swap
Opcode Decoder
Address Gen
Instruction Formatter
bs
opcode
reg_sadr
reg_dadr
ams
ind_adr
Internal Registers / Functions
Reg Mux Out
pcnt[11:0]
irq[3:0]
rst_n
clk
data[15:0]
wait
jump
State Machine
mdata_in[15:0]
mdata_out[15:0]
madr[11:0]
mrd
mwr
prd
hold
Memory Registers

Figure 2

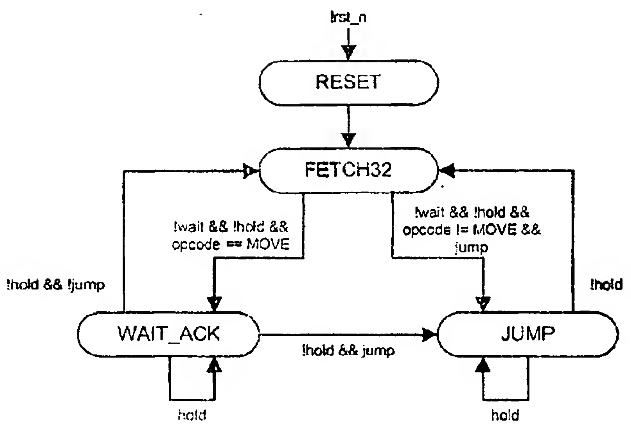

lrst_n
RESET
FETCH32
!wait && !hold && opcode == MOVE
!wait && !hold && opcode != MOVE && jump
!hold && !jump
!hold
WAIT_ACK
JUMP
!hold && jump
hold
hold

Figure 3

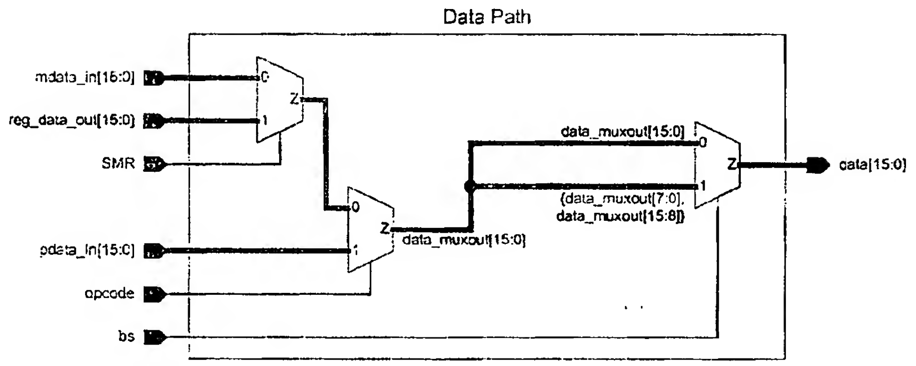


Figure 4

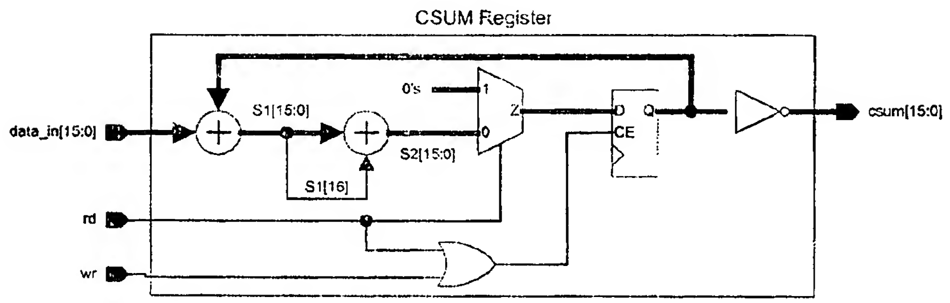


Figure 5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	DMR	Destination Address (DA)												DBS	x	x	SMR	Source Address (SA)												SBS

DMR	Selects the destination region ('0' for User Memory; '1' for Registers).
DA	Destination Address, in word.
DBS	The data word is byte swapped when DBS ^ SBS = 1.
SMR	Selects the source region ('0' for User Memory; '1' for Registers).
SBS	The data word is byte swapped when DBS ^ SBS = 1.

Figure 6

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	DMR	Destination Address (DA)												DBS	Constant Word (CW)															

DMR	Selects the destination region ('0' for User Memory; '1' for Registers).
DA	Destination Address, in word.
DBS	The data word is byte swapped when SBS = 1.
CW	Constant word to be loaded into the destination address.

Figure 7

REG_A	General Purpose Register A	
Access	Read/Write. Auto-Updated.	
Bit	Name	Description
B[15 :0]	A[15:0]	General Purpose Register A.

Figure 8

REG_B	General Purpose Register B	
Access	Read/Write. Auto-Updated.	
Bit	Name	Description
B[15 :0]	B[15:0]	General Purpose Register B.

Figure 9

PCNT	Program Counter	
Access	Write Only. Auto-Updated.	
Bit	Name	Description
B[15]	IE	If Equal jumping condition.
B[14]	IG	If Greater jumping condition.
B[13]	IN	If Not jumping condition.
B[12]	Reserved	Unused – always write "0".
B[11:0]	PCNT[11:0]	Program Counter. On a write, Program Counter is updated with PCNT[11:0] value if the following condition is met: case ({IE, IG}) 2'b00: 1 2'b01: IN ^ ((MASK & A) > (MASK & B)) 2'b10: IN ^ ((MASK & A) == (MASK & B)) 2'b11: IN ^ (((MASK & A) == (MASK & B)) || ((MASK & A) > (MASK & B))). After every program memory fetching, Program Counter is incremented by 1.

Figure 10

RETA	Return Register	
Access	Read Only. Auto-Updated.	
Bit	Name	Description
B[15:12]	Reserved	Unused.
B[11:0]	RETA[11:0]	Return Register. On a write in the Program Counter with IE = 0, IG = 0 and IN = 1, the Return Register is loaded with PCNT[11:0] value.

Figure 11

MASK	Mask Register	
Access	Read/Write.	
Bit	Name	Description
B[15 :0]	MASK[15:0]	Mask register used for some operations like comparaison, bit setting and bit clearing. By default this register is set to FFFFh.

Figure 12

WAIT	Wait Register	
Access	Read/Write. Auto-Updated.	
Bit	Name	Description
B[15]	T1	Set this bit to one makes the processor waiting for timer 1 reaches zero. If more than one bit is set in the WAIT register, the processor stops waiting on the first event. Read value is one when timer 1 is equal to zero.
B[14]	T0	Set this bit to one makes the processor waiting for timer 0 reaches zero. If more than one bit is set in the WAIT register, the processor stops waiting on the first event. Read value is one when timer 0 is equal to zero.
B[13:4]	Reserved	Unused – always write "0".
B[3:0]	I[3:0]	Set one bit to one makes the processor waiting for the corresponding interrupt. If more than one bit is set in the WAIT register, the processor stops waiting on the first event. Read this register shows which interrupt occurred and clears I[3:0] bits.

Figure 13

TIMER0	Timer 0 Register	
Access	Write Only. Auto-Updated.	
Bit	Name	Description
B[15 :0]	TIMER0[15.0]	Timer 0 Register. Write a non-zero value sets and starts the timer. It is decremented on every clock cycle. When it reaches zero, the timer is stopped and T0 bit is set in WAIT register.

Figure 14

TIMER1	Timer 1 Register	
Access	Write Only. Auto-Updated.	
Bit	Name	Description
B[15 :0]	TIMER1[15:0]	Timer 1 Register. Write a non-zero value sets and starts the timer. It is decremented on every clock cycle. When it reaches zero, the timer is stopped and T1 bit is set in WAIT register.

Figure 15

CSUM	CheckSum Adder Register	
Access	Read/Write.	
Bit	Name	Description
B[15 :0]	CSUM[15:0]	CheckSum Adder Register. On a write, the 16-bit one's complement sum is computed from the previous value and the written value. On a read, the read value is inverted (~CSUM[15:0]) and it is reset to zero.

Figure 16

DMA	DMA Register	
Access	Write Only. Auto-Updated.	
Bit	Name	Description
B[15:12]	Reserved	Unused – always write "0".
B[11:0]	LEN[11:0]	DMA Length Register. Write any value in the LEN field starts a DMA of LEN bytes (4096 if value = 0) from address contained in A[13:0] to address contained in B[13:0]. Bit 13 of each address register indicates if it's from/to register (0) or memory (1). During the DMA, memory addresses are incremented and register addresses are not incremented.

Figure 17

PCNT Flags			Comparators		Result	
E	G	IN	eq	gt	jump	call
0	0	0	X	X	1	0
0	0	1	X	X	1	1
0	1	IN	X	gt	IN ^ gt	0
1	0	IN	eq	X	IN ^ eq	0
1	1	IN	eq	gt	IN ^ (eq | gt)	0

Figure 18

Macro	Opcode	ArgA	ArgB	Description
JMP Addr[11:0]	LOAD	K_PCNT_ADR	{4'b0, Addr[11:0]}	Inconditional Jump at Addr[11:0].
JEQ Addr[11:0]	LOAD	K_PCNT_ADR	{4'b8, Addr[11:0]}	Jump at Addr[11:0] if (A & MASK) == (B & MASK).
JGT Addr[11:0]	LOAD	K_PCNT_ADR	{4'b4, Addr[11:0]}	Jump at Addr[11:0] if (A & MASK) > (B & MASK).
JGE Addr[11:0]	LOAD	K_PCNT_ADR	{4'bC, Addr[11:0]}	Jump at Addr[11:0] if (A & MASK) >= (B & MASK).
JNE Addr[11:0]	LOAD	K_PCNT_ADR	{4'bA, Addr[11:0]}	Jump at Addr[11:0] if (A & MASK) != (B & MASK).
JLE Addr[11:0]	LOAD	K_PCNT_ADR	{4'b6, Addr[11:0]}	Jump at Addr[11:0] if (A & MASK) <= (B & MASK).
JLT Addr[11:0]	LOAD	K_PCNT_ADR	{4'bE, Addr[11:0]}	Jump at Addr[11:0] if (A & MASK) < (B & MASK).
CALL Addr[11:0]	LOAD	K_PCNT_ADR	{4'b2, Addr[11:0]}	Inconditional Jump at Addr[11:0] and store PCNT register value in RET register. Used to call a sub-routine and return from it with the RET instruction.
RET	MOVE	K_PCNT_ADR	K_RETA_ADR	Inconditional Jump at address stored previously in RETA register. Used to return from a sub-routine CALL instruction.

Figure 19

Cycle	State Machine	Prog. Mem			Src Mem			Dst Mem		
	state	pcd	pcnt	pdata_in	re	sadr	smdata_in	we	dadr	dmdata_out
1	RESET	1	0	Invalid	0	x	invalid	0	Invalid	invalid
2	FETCH32	0	1	MV0 DA0,SA0	1	SA0	invalid	0	DA0	invalid
3	WAIT_ACK	1	1	MV0 DA0,SA0	0	x	RMD0	1	DA0	RMD0
4	FETCH32	0	2	MV1 DA1,SA1	1	SA1	invalid	0	DA1	Invalid
5	WAIT_ACK	1	2	MV1 DA1,SA1	0	x	RMD1	1	DA1	RMD1
6	FETCH32	1	3	LD2 DA2,K2	0	x	invalid	1	DA2	K2
7	FETCH32	1	4	LD3 DA3,K3	0	x	invalid	1	DA3	K3
8	FETCH32	0	5	MV4 DA4,SA4	1	SA4	invalid	0	DA4	Invalid
9	WAIT_ACK	1	5	MV4 DA4,SA4	0	x	RMD4	1	DA4	RMD4
10	FETCH32	x	6	JMP5 PA5	0	x	invalid	1	PCNT_A	PA5 = 20
11	JUMP	1	10	Invalid	0	x	Invalid	0	Invalid	Invalid
12	FETCH32	1	11	LD10 DA10,K10	0	x	invalid	1	DA10	K10
13	FETCH32	1	12	JEQ11 PA11 (F)	0	x	invalid	1	PCNT_A	PA11 = 6
14	FETCH32	0	13	MV12 DA12,SA12	1	SA12	invalid	0	DA12	Invalid
15	WAIT_ACK	1	13	MV12 DA12,SA12	0	x	RMD12	1	DA12	RMD12
16	FETCH32	x	14	JGT13 PA13 (T)	0	x	invalid	1	PCNT_A	PA13 = 6
17	JUMP	1	6	Invalid	0	x	Invalid	0	Invalid	Invalid
18	FETCH32	0	7	MV6 DA6,SA6	1	SA6	invalid	0	DA6	Invalid
19	WAIT_ACK	1	7	MV6 DA6,SA6	0	x	RMD6	1	DA6	RMD6
20	...									

Figure 20

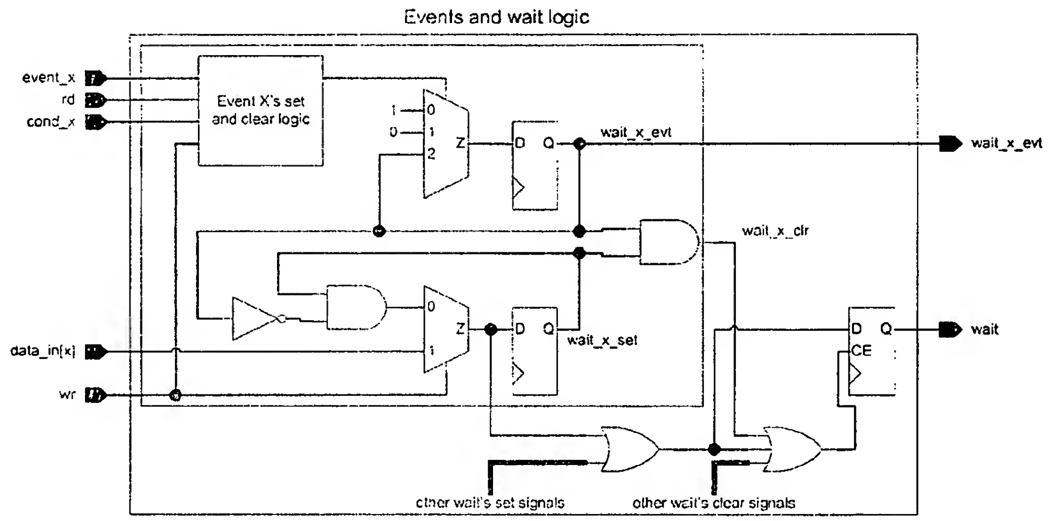


Figure 21

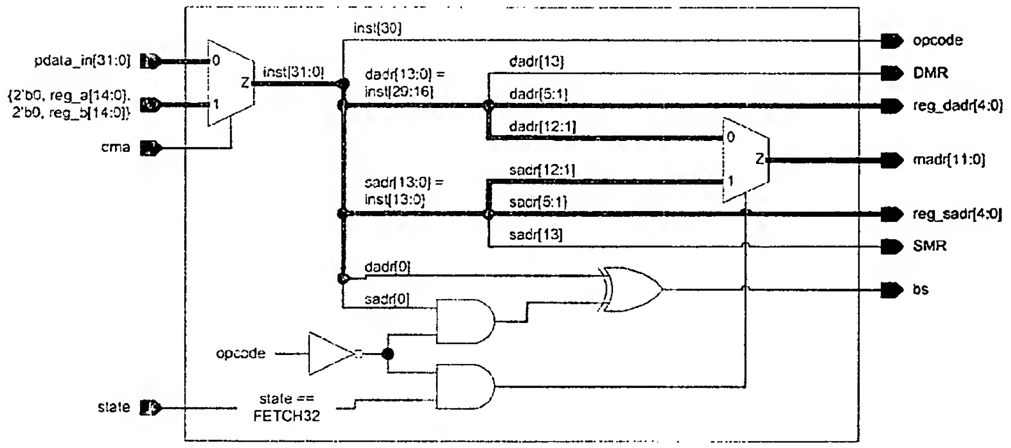
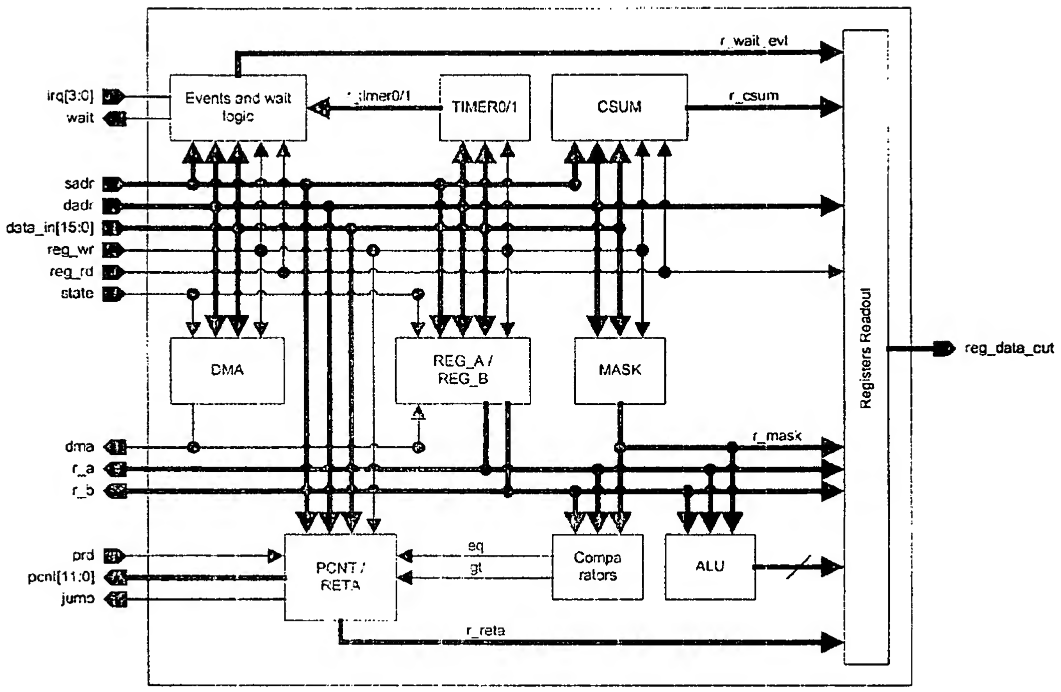


Figure 22


irq[3:0]
wait
sadr
dadr
data_in[15:0]
reg_wr
reg_rd
state
dma
r_a
r_b
prd
pcnt[11:0]
jump
Events and wait logic
r_timer0/1
TIMER0/1
CSUM
r_wait_evt
r_csum
DMA
REG_A / REG_B
MASK
r_mask
PCNT / RETA
eq
gt
Compa rators
ALU
r_reta
Registers Readout
reg_data_cut